0

AF INSTITUTE OF TECHNOLOGY

THE IMPACT OF IEEE-1076 ON VHDL

THESIS

Kevin J. Berk
Captain, USAF

AFIT/GCE/ENG/88D-1

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/88D-1

THE IMPACT OF IEEE-1076 ON VHDL

THESIS

Kevin J. Berk
Captain, USAF

AFIT/GCE/ENG/88D-1

AFIT/GCE/ENG/88D-1

THE IMPACT OF IEEE-1076 ON VHDL

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Kevin J. Berk, B.S. Physics, BSEE
Captain, USAF

December, 1988

Acknowledgments

I am very grateful to the many people who have helped my efforts on this research project. I would like to especially thank my thesis readers: Maj James W. Howatt who lent his expertise in the areas of language criteria, and the UNIX utility *yacc*; and Capt Bruce George who gave great words of encouragement throughout the whole project. I am also grateful for the guidance I received from my thesis sponsors, Dr. J. Hines and Lt N. Naclerio from the Air Force Wright Aeronautical Labs. To the simulator expert, Capt Douglas Pompilio, thanks for helping to find errors and for suggesting corrections. To the many members of my section, GCE-88D, who offered their friendship and words of encouragement, thanks, they came at the right time.

I would like to thank the two individuals most responsible for the completion of this thesis. Maj Joseph W. DeGroat, my thesis advisor, was always willing to take time from his busy schedule to discuss my ideas and answer my questions. He also gave me the necessary freedom and space to complete this research. Thanks for all your help. Finally to my wife[redacted] I could not have done it without your loving support especially during the times I felt quite discouraged. Your understanding and encouragement made the difference and helped to keep my thesis efforts in perspective. God Bless you always!

Kevin J. Berk

Table of Contents

Page

Page

List of Figures

List of Tables

AFIT/GCE/ENG/88D-1

Abstract

This paper describes the conversion and extension of the Air Force Institute of Technology's (AFIT's) UNIX-based VHDL Analyzer. This task concentrated on converting the current AFIT analyzer from VHDL Version 7.2 to IEEE Standard 1076-1987 and extending it to include the *wait statement* and *component instantiation* as defined in IEEE Standard 1076. An evaluation was also performed on the differences between Version 7.2 and IEEE Standard 1076-1987 using nine predefined criteria that determine if a programming language is good. The evaluation was done to determine if IEEE Standard 1076-1987 was indeed a better version of VHDL than its predecessor, Version 7.2. Results from validation testing showed that the IEEE-1076 analyzer implements 60 to 70% of IEEE Standard 1076 and efficiently used computer resources.

THE IMPACT OF IEEE-1076 ON VHDL

I. Problem Statement

1.1 **Problem**

The IEEE has recently approved a new standard, IEEE Standard 1076-1987 [IEEE88] (which will be referred to hereafter as IEEE-1076), for the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL). IEEE-1076 differs from the previous revision of VHDL, Version 7.2 [Inte85]. The set of changes and additions contained in IEEE-1076 appear to be better, but have not been subjected to any evaluation to determine if IEEE-1076 is indeed a better version of VHDL. To gain knowledge about VHDL and determine the impact of IEEE-1076 in the UNIX[1] environment, the two versions of VHDL need to be evaluated using criteria (which will be defined in Chapter 2) and aided by converting a current VHDL analyzer (which syntactically and semantically analyzes VHDL source code and outputs the semantic information in an intermediate form) to the IEEE-1076 standard.

[1]UNIX is a trademark of AT&T.

1.2 Background

Because many incompatible hardware description languages (HDLs) exist, the Department of Defense (DoD), to promote VHSIC technology transfer, in 1985 established a standard, the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) [Dewe86:12] [Int84a:9-1]. A VMS [Deit84:507-508] operating system-based tool kit, including an analyzer [Int84a], was delivered with the DoD standard. Concurrently, however, the Air Force Wright Aeronautical Laboratories (AFWAL) and the Air Force Institute of Technology (AFIT) felt that to encourage universities to perform VHDL research a UNIX-based tool set should be made available to them at little or no cost. This tool set became known as the AFIT VHDL Environment (AVE) [Cart87:3]. *Consequently*, in 1986, Capt Frauenfelder, an AFIT student, developed a prototype UNIX-based VHDL analyzer [Frau86]. This product was expanded in 1987 by another AFIT student, Capt Bratton, into a subset analyzer [Brat87:6.2] that incorporated 75% of VHDL Version 7.2 [Inte85], the current version of VHDL at the time for the DoD.

During the development of VHDL, representatives from government, industry, and academia participated in the review process. Improvements were suggested by the participants, many of which were incorporated into the VHDL language. Since the final product of this development, Version 7.2, received a favorable response [Aylo86:26-27], the IEEE began the process of creating an industry-wide VHDL standard using Version 7.2 as a baseline. The result of the standardization effort of

VHDL was IEEE Standard 1076, a new industry-wide standard for VHDL. Since IEEE-1076 has been recently published (March 1988) and contains many changes and additions to the baseline Version 7.2, no evaluation using any predefined criteria has been performed to determine if the new standard is actually better than Version 7.2.

1.3 Scope

The goal of this project was to determine if IEEE-1076 is a better version of VHDL by evaluating the changes and additions to Version 7.2. This was done in conjunction with converting a UNIX-based Version 7.2 analyzer to an IEEE-1076 VHDL analyzer.

More precisely, the goals required evaluation criteria that were predefined, and an existing VHDL Version 7.2 analyzer that incorporated either a subset of VHDL Version 7.2 or the whole language.

The criteria used for the evaluations should be defined prior to use so as to keep the evaluator's bias to a minimum. The criteria should also have been used in previous evaluations and have an established and proven record.

The analyzer should be a UNIX-based analyzer implementing the whole language or a subset thereof. Bratton's analyzer was an ideal starting point. He was able to implement 75% of Version 7.2 [Brat87:6.2] which provided a basis for comparison of a subset of VHDL. While Bratton's analyzer was being revised the differences between version 7.2 and IEEE-1076 were studied to determine whether they added to or detracted from the language as a whole. Revising and extending an analyzer aided in studying the new constructs in detail because both the old and new had to be learned in order to compare them.

Many of the same requirements that Bratton imposed [Brat87:1.7] on the analyzer project are still applicable for this project. The purpose of the analyzer remains the same: it is to check the source code input for correct syntax and semantics and output the semantic information in a machine readable form.

The IEEE-1076 analyzer will also retain the qualities of generalization, reliability, maintainability, and economy as applied to the analyzer project by Bratton [Brat87:1.7-1.11]. They are reviewed briefly below.

Generalization means that the analyzer must be UNIX-based and be able to use the tools that are common to that operating system such as *lex* [Lesk78], *yacc* [John78] and the programming language C [Kern78, Kern88]. The analyzer must also be *portable*; be able to operate on several different computer configurations. The analyzer must also handle correct and incorrect input without "crashing".

For the analyzer to be *reliable*, it must pass a comprehensive set of tests with and without the other tools of the AVE.

To be *maintainable* the analyzer must be well documented, and designed in a structured and modular fashion.

To be *economical*, the analyzer should be efficient when analyzing designs. Bratton's goal was "1000 lines of source code per CPU minute on an unloaded VAX 11/780 running bsd 4.2" [Brat87:1.11]. Main memory should also be conserved with a goal of 640K of main memory for any design and 100 bytes of VIA output for each input statement [Brat87:1.11].

Since the focus of this project was to determine if the IEEE-1076 is a better version of VHDL than Version 7.2, extensions to the converted analyzer were limited to adding the *wait statement* and the new definition (non-primative) for *component instantiation.*

The *wait statement* enhances the simulation capability of the language [Nash86:59] and eases the burden on the user to design the same capability without the *wait statement.* The *wait statement* replaces several statements in order to perform the same function.

Component instantiation was chosen because of its "nested hierarchy of block statements" [CLSI87:11-3]. It allows many levels of components to be instantiated making for a more hierarchical design.

The following related subjects however, were not included in the scope.

Incorporation of the whole language was not part of the scope of this project since the focus was on the comparison between Version 7.2 and IEEE-1076. The conversion of the analyzer from Version 7.2 to IEEE-1076 aided in performing this evaluation.

Optimization of the VHDL Intermediate Access (VIA) was not included in the scope of this project either. Since work was concurrently being performed on the AVE simulator [Pomp88], it was crucial to the success of both projects to maintain the VIA in its stablest form. Changes were made to the VIA only to accommodate changes in the language definition.

1.4 Approach

To meet the requirements of the language evaluation and the analyzer conversion and extension, the following tasks had to be performed.

1. *Find predefined criteria.* As mentioned in the scope, it is necessary to use predefined criteria to perform the evaluation to try to minimize the evaluator's bias and to prevent the shaping of criteria to meet the results of the evaluation. The criteria description is presented in Chapter 2.

2. *Compare the two versions of VHDL.* This task was composed of two parts. The first was to evaluate the changes and additions of IEEE-1076 against the criteria, and the second part was to determine what constructs needed to be changed both syntactically and semantically.

3. *Revise and extend the current analyzer.* After the differences were determined, the analyzer was changed to syntactically accept the full IEEE-1076 language. The subset of the language that was semantically analyzed was extended to add the *wait statement* and the new definition of *component instantiation.*

4. *Develop or obtain a validation test suite for the analyzer.* The Intermetrics VHDL analyzer test suite [Int84b] was converted to IEEE-1076 by Steve Grout a subscriber to the VHDL repository. The converted test suite is available from the VHDL repository.[2] This test suite covered most of the IEEE-1076 language. Test cases were developed to test the areas not covered by this test suite.

5. *Validate the changes and extensions.* The changes and extensions were validated using the test suite previously described. Regression testing was performed after each change or set of changes were implemented to ensure that no other constructs were inadvertently changed or affected. The results of the validation are presented in Chapter 6.

6. *Perform analyzer-simulator integration tests.* The AVE simulator [Pomp88] was being converted to handle IEEE-1076 designs at the same time as the analyzer was being converted. At several points during the analyzer conversion, integration tests were performed to show end-to-end results. A circuit could be modeled in a mix of Version 7.2 and the new IEEE-1076 and be simulated in this intermediate configuration.

7. *Document the results.* This thesis contains the decisions and conclusions of this project.

[2]The arpanet address of the VHDL repository is WSMR-SIMTEL20.ARMY.MIL

1.5 Maximum Expected Gain

If this country is to stay technologically competitive in integrated circuitry, VHSIC research needs to be performed both in industry and in universities. Understanding the differences between Version 7.2 and IEEE-1076 when based on predefined criteria is important to understand the direction that the language has taken. Future changes to the language can be better made in light of this evaluation. A UNIX-based (the *de facto* university standard) IEEE-1076 Analyzer will help to further VHDL research and VHSIC technology insertion into the academic community.

1.6 Overview of the Thesis

This thesis is presented in seven chapters. Chapter 2 shows the results of a literature review of past and ongoing VHDL research. Chapter 3 describes the management of the project, Chapter 4 presents the language evaluation, and Chapter 5 shows the analyzer conversion and extension. Thesis validation will be discussed in chapter 6. The findings and conclusions along with recommendations for future research efforts will be presented in chapter 7.

II. Literature Review

2.1 Introduction

As stated in Chapter 1, the primary goal of this thesis was to determine if IEEE-1076 is better than Version 7.2 for defining VHDL. This was aided by converting an existing analyzer from Version 7.2 to IEEE-1076. The revised analyzer syntactically and semantically checks the VHDL code written in accordance with IEEE Standard 1076 and outputs the results in an intermediate form. The intermediate form is then used by other tools of the AFIT VHDL Environment (AVE) [DeGr88] for such activities as simulation. By using VHDL, chip and circuit designers can describe their designs in software and have a computer analyze and simulate them before actual fabrication.

This literature review will concentrate on the areas that pertain to this thesis: standardization of hardware description languages (HDLs), VHDL, and previous research into VHDL analyzers to provide a brief history and background. Also covered are programming language evaluation criteria and software project management for both software maintenance and new software development.

2.2 Automated Design Standards

Since many innovations are being developed in automated design (which include HDLs), it is difficult to communicate and transport designs within and especially between companies. International corporations face even more difficulties transporting designs across national boundaries [Radk88:48-49]. The academic community, government, industry are all motivated to standardize automated design. Students would be able to spend time on learning material rather than on learning different tool formats and be better prepared for industry when they graduate. The government needs clear specifications for its contracts. Vendors can become more competitive with each other, keeping prices lower, and buyers will have to bear less risk from a single supplier going out of business. Designs can be reused when upgrading tools.

"Without structure to a problem definition, we have total anarchy and huge inefficiencies" [Radk88:53]. Standards are available even if only *de facto* and they need to be used and worked on so that improvements can be made to them. Standards are needed by customers and vendors even at the risk of restricting innovation slightly in the short run [Radk88:55].

2.3 The VHDL Language

The need to design and implement VHSIC technology for military systems was a major incentive in the development of VHDL. "Moreover, VHDL will improve documentation and decrease design time and cost for government electronic systems" [Dewe86:12].

To determine the requirements for VHDL, many HDLs were analyzed by the Institute for Defense Analysis in 1981 to determine and make use of the HDL's advantages [Aylo86:17]. The major features selected were:

> scope of the hardware, management of the design, timing description, architectural description, description of a design's interface, descriptions of a design's environment and language extensibility [Aylo86:18].

Some features such as language extensibility are not supported in VHDL. Syntactic and semantic extensions cannot be made to VHDL. [Aylo86:26].

The VHDL language is not limited to any one technology such as CMOS or bipolar but is technology independent. It is a flexible tool able to describe various levels of abstraction from an overall system down to "the logic gate level" [Lips86:28].

In 1983, Intermetrics was awarded a contract to develop a VHDL tool set for the U.S. Air Force [ASD83]. The tool set was developed under the VMS operating system. Since the academic community primarily operates its computer resources under the UNIX operating system, the Intermetrics toolset is not available for research and classroom support computers. The AFIT VHDL Environment (AVE), which includes both an analyzer and a simulator, was developed in response to this situation [Cart87].

Recently though, the IEEE has developed an international VHDL standard, IEEE-1076 as stated in Chapter 1.

2.4 Previous Research

Building a VHDL analyzer has been the topic of two previous theses at AFIT. In 1986, Capt Frauenfelder developed a UNIX-based prototype analyzer [Frau86], implemented in the C language, that incorporated a subset of VHDL Version 7.2 [Inte85]. Capt Frauenfelder's prototype analyzer was able to process "over 1/3 of the VHDL productions" [Frau86:5.14].

Capt Bratton continued the research effort on the prototype analyzer at AFIT. His objective was to build a "production quality" analyzer [Brat87] for which he defined the *production quality* requirements and then tested the analyzer to show that the requirements were met. Bratton's efforts were also devoted to developing a new and more efficient VHDL Intermediate Access (VIA). He was able to implement 75% of Version 7.2 [Brat87:6.2].

Research into analyzing VHDL is also being performed at other universities in this country. L.A. Mears and S.P. Levitan at the University of Pittsburgh, are researching capture and simulation tools [Mear88]. Their efforts also include a compiler that takes the VHDL input and outputs an intermediate form which is a flattened circuit description in a text file. The compiler itself was built using the UNIX utilities *lex* [Lesk78] and *yacc* [John78]. The design of the compiler is based on Capt Frauenfelder's prototype analyzer [Frau86] and the *SampleC* compiler [Schr85]

At the University of Cincinnati, under the direction of Dr. Harold Carter, analyzer research is being done in the UNIX environment. While an analyzer is being built by one of his students, research efforts are also being performed in other areas of VHDL such as synthesis behavioral structure semantics and simulation in the high-speed parallel environment [Cart88].

VHDL is not only being studied in the research labs, but is making its way into the classroom. The University of Virginia is now offering a design course in VHDL at the undergraduate level, one of the first in the country [Weis88:51].

2.5 Language Evaluation Criteria

In order to perform a fair evaluation of the changes and additions to VHDL, criteria needed to be found or developed and predefined. High order language criteria development had not been formalized until the Ada[1] language was being defined by the DoD in the late 1970's. The *STEELMAN* document [STEE78] contains the technical requirements or criteria that a high order language should meet. This document contained very detailed requirements. Tucker subsequently distilled these requirements down to nine more general requirements or criteria that a good programming language should have [Tuck86:10]. The nine criteria determined by Tucker are: expressivity, well-definedness, data types and structures, modularity, IO facilities, portability, efficiency, pedagogy, and generality. The definitions of each follow.

Expressivity is defined as the expressiveness of the language, or the clarity of the meaning intended by the programmer. It is also defined as the compactness of the

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

program and how well the language encourages structured forms of programming. Uniformity of the meaning of the symbols that are used throughout the program is also a factor in determining expressivity.

Well-definedness indicates to what degree the syntax and semantics are free of ambiguity, are consistent and complete throughout the language. It also indicates how well the behavior of the language can be predicted before it is used.

The *Data Types and Structures* criterion determines if the language supports a variety of types, nonelementary collections of types (records), and dynamic data structures; those declared "on the fly".

Modularity is defined in two parts: the ability to support subprogramming, and extensibility of the language allowing the programmer to define operators and data types.

The *IO Facilities* criterion refers to support for sequential, indexed, and random access to disk files.

Portability simply refers to how machine-independent the language is defined.

The *Efficiency* criterion looks at how fast a program compiles and how easy it is for a programmer to develop a program.

Pedagogy is defined as the ease of teaching and learning a language. Well-written documentation and non-confusing constructs of a language are desired.

Generality refers to the range of programming applications for which the language is well-suited.

If a programming language, whether targeted for software or hardware, favorably meets all of the above criteria, then the language is considered good. Further discussion and use of these criteria will be presented in Chapter 4.

2.6 Software Project Management

There are many software development methodologies available to use for both software maintenance and enhancement. Several models were considered for use on the project. Five models are presented.

2.6.1 Code and Fix Model This model was very basic only containing two steps: write some code and fix any problems in the code. This was done iteratively until the code performed in the prescribed manner [Boeh88:61].

2.6.2 Stagewise and Waterfall Models The stagewise model determined that "software be developed in successive stages (operational plan, operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, system evaluation)." [Boeh88:63]. The waterfall model expanded on it to add feedback loops to the previous stage and rapid prototyping during the requirements phase.

2.6.3 Evolutionary Development Model This model starts with a quick operational program and builds on it as the requirements evolve. It is used when the requirements are not known and the user does not quite know how to specify the requirements [Boeh88].

2.6.4 Transform Model This model automatically transforms a specification into code. This is used when the specification is known and understood upfront, and automatic tools are available to perform the transform [Boeh88].

2.6.5 Spiral Model The spiral model is a flexible model. It can accommodate most other models including the aforementioned. This spiral model breaks the effort into smaller pieces (or spiral), each piece being completed before starting the next [Boeh88]. A spiral is make up of of many cycles (the number is arbitrary and usually determined by the complexity of the project). Each cycle undergoes about four distinct steps. The first is to define the objectives of the product (or a particular part of the product) being developed, the alternate means of implementation and any constraints imposed. The second step identifies areas that could pose significant risk to the project. Once the risk is identified the third step determines the strategy to resolve this risk. This could take many different forms including rapid prototyping, benchmarking, modeling, . . etc. Once the risks are resolved the product for this cycle is developed and verified. The last step involves planning for the next spiral and ends in a review of the just-completed cycle.

2.7 Summary

This chapter presented a brief overview of design automation standardization, the VHDL language, previous research, language comparison, and software management models. The following chapters will show how the previous research has been used to aid in evaluating the impact of changing from Version 7.2 to IEEE-1076 using the described criteria. It will also provide the overall system design alternatives and decisions for converting and extending the existing analyzer.

III. Management of the Project

3.1 Overview

This chapter describes the overall management of the project to solve the thesis problem as stated in Chapter 1. The software project management models that were presented in Chapter 2 will be briefly discussed and compared in order to choose one to use on the project.

3.2 Management of the Software Project

All software projects need to be planned and properly managed to decrease the risk of failure. Several models for software project management are available for use, some of which were described in Chapter 2. Further discussion of these models follows.

The *Code and Fix Model* is inappropriate to use since it does not follow an overall design, but rather builds around whatever happens to exist or was started in the beginning of the project. The results are poorly structured code and expensive subsequent fixes [Boeh88:62]. This model will not work for management of this project since the overall design is known and already modular.

The *Stagewise and Waterfall Models* have been the most popular and are required for DoD contracts [DoD88:10]. Even though the model provides for feedback to the preceding step, errors that creep into the project in the beginning, that are not detected in the succeeding step, propagate into the later stages of the project where they are more difficult and more expensive to fix. For this reason, this model was rejected for use.

The *Evolutionary Development Model* is useful when the requirements for the project are not well-known. The user may not know what the final product should be like until it is seen. In this project, however, the requirements are well-known and not subject to change during the life of the project. This model was rejected.

The *Transform Model* which automatically transforms a specification into code would be very useful, but the tools do not exist to perform the transformation for this project except for the UNIX utilities which build the lexical analyzer and the parser. This method was rejected for managing the whole project.

The *Spiral Model*, however, provides a flexible and user customizable model. It emphasizes that the risks of the project be defined prior to continuation. Errors are more easily caught in the beginning phases because of the highlighting of the risks. The number of "spirals" is not limited, but can contain as many as needed. This insures that the project is divided into small definable pieces and that each risk is resolved before moving on in the project. This model was chosen for the project also because it is flexible enough to be used on a maintenance type project (conversion of the Version 7.2 analyzer to an IEEE-1076 analyzer) and new development (extending the capability of the converted analyzer). The specific steps of how the spiral model was used can be found in chapter 5.

3.3 Management of the Version Evaluation

Management of the evaluation was actually a simplified version of the spiral model. Since this was mostly a side-by-side comparison of documents with some experience from the analyzer, there was very low risk in performing the analysis. Also, the possibility of errors propagating into the next phase was not a problem.

The evaluation was managed and performed independently of the conversion-extension of the analyzer. The evaluation was not tied to any "milestones" of any schedule, or to any of the specific steps of the analyzer conversion-extension. However, the two tasks mutually aided each other and in reality were performed in parallel.

3.4 Summary

The overall management of the project was discussed in this chapter forming the framework of how the actual work was accomplished for both the version evaluation and the software portion (the analyzer conversion-extension). Chapters 4 and 5 detail the specific design and steps used within this management framework.

IV. VHDL Version Evaluation

4.1 Introduction

This chapter presents the details of the two tasks presented in Chapter 1: *find predefined criteria* and *compare the two version of VHDL*. First the criteria is determined and then the documents used for the evaluation are identified. Next, the specific steps used to perform the evaluation are enumerated and described followed by the actual results of the evaluation.

4.2 Find predefined criteria.

In order to perform an evaluation of programming languages that is as unbiased as possible, criteria are needed that are defined before the actual evaluation takes place. Hopefully these criteria which are already defined, are also applicable for the specific evaluation, and have been used successfully in a previous evaluation.

The *STEELMAN* document [STEE78], one of several in a series written while Ada was being developed, contained very specific and detailed requirements that the DoD standard language should meet. Several languages were evaluated against the criteria to determine if any were suitable to become the DOD's standard language.

The criteria in the *STEELMAN* document were too numerous and detailed to use on this project. However, they served as a basis for evaluations and comparisons of several general purpose programming languages [Tuck86:10-12]. Tucker distilled the *STEELMAN* requirements down to nine criteria that he used to "determine if a computer programming language was good." [Tuck86:10]. These nine criteria were described in Chapter 2.

Tucker's nine criteria were chosen to be used in this project. They are predefined and have been used to make several comparisons. These criteria were at a level high enough to allow differentiation between programming languages (or in this project's case, versions) and broad enough to cover the whole breadth of the programming language capabilities. The *STEELMAN* criteria would have caused the project to be bogged down in minute detail, which was not within the scope of the project.

4.3 Compare the two versions of VHDL.

Three documents were used for this evaluation: The *VHDL Language Refinement Rationale* [CLSI87], the *VHDL Language Reference Manual Version 7.2* [Inte85], and the *IEEE Standard VHDL Language Reference Manual* [IEEE88]. The results of this evaluation can be found later in this chapter.

4.4 Evaluate Version Differences

Several steps were followed to perform the version difference evaluation. The steps are shown below:

1. Determine or define criteria.

2. Determine differences both syntactic and semantic.

3. Match the differences to one or more criteria.

4. Determine whether the effect of the change enhances or detracts.

5. Document the results.

1. Determine or define criteria. The criteria was chosen earlier in this chapter and is described in Chapter 2. This step was performed only once while steps two through five were performed iteratively.

2. Determine differences, both syntactic and semantic. The differences were determined in a very straight-forward manner. The Version 7.2 LRM and the IEEE-1076 LRM were compared in a side-by-side comparison. The *Language Refinement Rationale* was also heavily consulted for each difference detected.

3. Match the difference to one or more criteria. After the difference was detected, it was matched against the criteria to determine if it would fit under one or more of the criteria. The evaluator's opinion play d a part in this step, but was minimized since the criteria were predefined and there was very little overlap among the criteria.

4. Determine whether the effect of the change enhances or detracts. This was a relatively easy task. Changes to a language are usually made to the betterment of the language, but since IEEE-1076 was designed by committee this was felt to be a necessary step. Each difference was weighed against the definition of the particular criterion to determine whether it added to or detracted from the VHDL language.

5. *Document the results.* The detailed results are documented below.

4.5 Evaluation Details

The following are the detailed results of the differences between Version 7.2 and IEEE-1076 to determine if IEEE-1076 is indeed a better version of VHDL. The results are presented per criteria as described in Chapter 2.

4.5.1 **Expressivity** The expressivity of VHDL was enhanced by several changes and additions in IEEE-1076.

- The new design entity concept is simpler and hence makes the language more expressive from a programmer's view point. "A design entity is defined by an *entity declaration* together with a corresponding *architecture body*" [IEEE88:1-1] instead of *all* of the corresponding architectures using the entity as defined in Version 7.2. The design hierarchy changed from one of design entities to that of blocks. Blocks now represent the design hierarchy and design entities represent source code hierarchy. This new design concept increases the structuredness of the program.
- Signal declarations are now allowed in packages so they can be global. Import directives were deleted as a result of the global signal declarations, making the language more compact.

- The alias declaration was clarified. It is now an alternate name for an object, not another object.
- IEEE-1076 clarifies labels by allowing both indexed labels and label declarations. This will allow differentiation between generated statements. Indexed labels provide unique names for instances of components or blocks.
- The *nand* and *nor* operations are not associative by definition. The the change to the IEEE-1076 expression syntax corrected this.

Conclusion:

IEEE-1076 is more expressive than Version 7.2. All of the above differences have a positive impact on VHDL.

4.5.2 **Well-Definedness** Several changes were made to VHDL to further define the language and eliminate some confusions.

- The new design entities enhance the definition of VHDL by making the syntax and semantics more consistent. "Blocks, entities, and architectures now all have header, declarative parts, and statement parts" [CLSI87:1-2]. The syntax for declaring ports and generics is also common. A conceptual change in the design hierarchy was made to VHDL; a design entity is defined as only one architecture and one entity declaration.
- The keyword **begin** is now used by entities, architectures, and blocks to maintain syntax consistency.

- Generic and port lists are more consistently used. They appear in the same order in entity declarations, blocks, component declarations and instantiations, and binding indications.

- The syntax for configuration declarations was changed to differentiate between configurations for blocks and components. Previously, configuration body declarations could be confused with configuration specifications.

- IEEE-1076 specifies minimum ranges for the integer, floating point, and physical type Time types for a more complete definition of these types.

- The **atomic** type was removed from IEEE-1076 since it was asymmetric in Version 7.2. A composite signal had to be assigned as a whole while its subelements were allowed to be read independently.

- Composite types are redefined in IEEE-1076 as a collection of scalar types vice the Version 7.2 definition which allowed collections of **atomic** types or the **atomic** objects themselves.

- A constrained array type is defined in IEEE-1076 by using an anonymous base type as Ada does so types can be cleanly defined.

- The matching elements concept as defined by IEEE-1076 was included to help clarify correspondences "between arrays, in associations, aliases, array conversions, and logical and relational operations" [CLSI87:5-8]. This allows the passing of actual parameters with the same size but different bounds.

- Default values are now defined to support deterministic simulation. They are used when signals are not explicitly defined.
- IEEE-1076 uses the reserved word **bus** and a resolution function to resolve bus signals.
- Two signal kinds were defined and added to IEEE-1076: **buses** and **registers**.
- The syntax for component declarations is more consistent with that of blocks and entities. Default expressions are allowed on IN ports, and extra syntax (semicolon) was added after **port** and **generic** clauses.
- The declarative parts of VHDL were redefined so that IEEE-1076 was more orthogonal; separate concerns are defined separately.
- Specification syntax is more consistent in IEEE-1076 than in Version 7.2. Each specification has a unique reserved word, "the name of the items specified, (or **others** or **all**), a colon, the type, entity class, or component type of the items, a delimiting reserved word, and the information associated with the item." [CLSI87:7-4].
- Physical type operations are more explicitly defined. The physical type operations for the underlying integer computations have a clearer relation with the range limits and precision of a particular implementation.
- Mixed type arithmetic operations were changed due to overloading and name-hiding rules. Exponentiation can be performed by predefined **Integer** only,

and mixed physical and integer operations must specify type **Integer**, not just any integer type.

- Local static, global static, and dynamic expressions are defined making the classification of expressions clearer.
- Aggregates can now be the target of a signal or variable statement. In Version 7.2, an aggregate could be assigned to a composite object but a composite object could not be assigned to an aggregate. This resulted in an asymmetric situation.
- Signal assignment now has no multiple-signal target, only a single name or an aggregate is allowed. Version 7.2 confusingly implied that the multiple targets were tied together.
- The ":=" of the loop statement was changed to **in** to match the syntax of generate statement. The loop and generate statements are similar in function.
- The block syntax was changed to follow the new concept of design entities. The Guard function also changed so that only blocks that contained a guard expression were guarded.
- The concurrent signal assignment statement syntax was changed to match the sequential signal assignment statement syntax. The reserved word **memoried** was changed to **guard** for clarity.
- Component instantiation statement syntactic changes followed those in blocks and entities for consistency.

- The scope rules were redefined to better define control and visibility. Entities and architectures are one declarative region and the scope of blocks extend into a block configuration.

- The **use** clause has been better defined (made selective) to allow finer control over which names can be made directly visible by allowing them in the declarative regions.

- The definition of design units was simplified into a more logical grouping. Interface declarations were renamed as entity declarations and architecture body declarations were renamed as architecture declarations. The number of design units were decreased from 6 in Version 7.2 to 4 in IEEE-1076. Subprograms are no longer design units. Configuration body declarations were also changed since there was confusion in Version 7.2 between configuration bodies and architecture bodies.

- The **use** clause is now selective and distributive. The **with** clause of Version 7.2 was removed since the **use** clause now contains the with clause functions. The syntax was simplified by removing the reserved words **procedure, function,** and **package** since they added nothing to the understanding of the construct and duplicate design unit names are not allowed.

- Elaboration of all declarations and statements are defined in detail in an effort to alleviate the confusion and complexity which resulted from not specifying the elaboration in Version 7.2.

- The predefined attributes 'Stable and 'Quiet were redefined to sense delta-length pulses or oscillations. Version 7.2's definition could not detect them.

Conclusion:

This criterion showed the most improvement by far. The syntax of IEEE-1076 is much more consistent than that of Version 7.2. All of the above differences have a positive impact on VHDL.

4.5.3 **Types** Several new types were added to IEEE-1076.

- Version 7.2 allowed a variety of data types, but did not support dynamic data structures. IEEE-1076 adds the **access** type which provides for dynamic data structures. Trees, stacks, and queues can now be abstractly supported.
- The file type was added to ease the communication with external files.
- Null arrays were added.
- Numeric (real to integer) and array type conversions (array to array of similar types) were added. Type conversions were difficult or impossible to do in Version 7.2.

Conclusion:

IEEE-1076 is improved over Version 7.2 in this category by the addition of new capabilities. All of the above differences have a positive impact on VHDL.

4.5.4 **Modularity** The modularity of VHDL has been enhanced by several changes.

- Subprograms are allowed in any declarative region putting them closer to where they are actually used. Previously they were only allowed in packages.
- Separate subprogram declarations and bodies isolate each from the other allowing indirect recursion. However, bodies of subprogram declarations must be in the body of the package in which it is declared.
- Subprogram and predefined operator overloading now allows the user to redefine objects which provides for more extensibility of the language.

Conclusion:

Only three changes were made under this criterion, but they were all positive. Even though subprogramming itself wasn't changed, the added flexibility of where subprograms can be placed added to the modularity part of the criterion. All of the above differences have a positive impact on VHDL.

4.5.5 **IO Facilities** VHDL version 7.2 did not support any general I/O capability in its environment, but IEEE-1076 does with the addition of File IO.

- File IO however, is restricted to sequential files to avoid sneak paths between internal processes. The mode of a file is restricted to either **in** or **out**. Four operations are supported: Read, Write, Empty, and Full.

- Internal files process inter-process communications and allows modeling of a fixed size queue.
- IEEE-1076 provides package **TextIO** for formatting text File IO.

Conclusion:

Since Version 7.2 did not support any I/O, adding File I/O even even though it is restricted to sequential files, is an improvement to VHDL. All of the above differences have a positive impact on VHDL.

4.5.6 **Portability** At least two decisions on whether or not to include certain constructs were based on portability.

- In order to maintain portability, IEEE-1076 does not specify how real numbers should be rounded since this is hardware dependent. The rounding direction on some hardware is unknown or undeterministic.
- Design library redefinition enhances portability. A non-hierarchical library can be more easily supported by more computers than a hierarchical library. It can also support multiple parts libraries from different vendors.

Conclusion:

Both of the above differences have a positive impact on VHDL.

4.5.7 **Efficiency** The efficiency of VHDL is improved in IEEE-1076. Several changes enhance the ability of large circuits to be designed by teams.

- Packages now have separate declarations and bodies to ease the task of team design and ease the development of programs.
- IEEE-1076 also allows a package declaration by itself so that code written in another language can be used (or reused) as the package body.
- There is no longer a need to explicitly declare a subtype since subtype indications are allowed in the interfaces.
- The addition of the concurrent procedure call statement was added to make timing checks easier.
- Nonstructural, multiple libraries are allowed in IEEE-1076 allowing faster access.
- Revisions of a design library are no longer supported. They reflected analysis numbers not hardware revision numbers. They were also inconvenient to change in the source code.
- The attributes 'LeftOf and 'RightOf were added for symmetry for dynamic subranges and unconstrained arrays. The attribute 'Reverse_Range allows the stepping backwards through a range. The user now does not need to provide these capabilities.

Conclusion:

VHDL's efficiency has improved in IEEE-1076. Large designs are more easily divided with the introduction of separate package declarations and bodies. Design library access should be faster in a nonstructural library. All of the above differences have a positive impact on VHDL.

4.5.8 **Pedagogy** Several changes were made to the language which directly and indirectly affected the pedagogy of the VHDL language.

- Several changes were made to make the syntax consistent throughout the language. A language is easier to teach and learn if the syntax is consistent throughout the language.
- The LRM is written in a hierarchical manner with a few examples of how the constructs should be used.
- The **import** and **select** directives of 7.2 were removed because they complicated the scope rules and semantics of default configurations respectively.
- Subprograms are no longer design units since hardware design units are primary in VHDL. Subprograms are considered software design units. This presents less confusion in the hierarchy of hardware design.
- Attributes are now explicitly associated with their items and are no longer inherited from objects of a certain class.
- Removal of **enable** and **disable** statements help the ease of learning the language. They were difficult to decipher since they had complicated interactions with process sensitivity lists.

Conclusion:

So far all the changes made have made VHDL a better language. Better definitions, more consistent syntax and the removal of statements that added more complications than capabilities improved the pedagogy of VHDL. Missing from the available documentation is a thorough tutorial and programmer's manual. The remaining differences above have a positive impact on VHDL.

4.5.9 **Generality** The capability of VHDL to describe and model circuits has been increased to include more types of circuits.

- Structural recursion is now allowed so that more and diverse circuits can be defined.
- System level modeling of stacks, trees, queues, and other abstract data types is now supported by IEEE-1076 since access types were added.
- Added signal kinds of **bus** and **register** make it more convenient to model these kinds of hardware.
- The port drive requirements were relaxed to take a more global or network view of the circuitry.
- Independent formal subelement associations allow elements of a composite formal to be associated to separate objects.
- Global signals are allowed in packages to ease the design of power and ground lines.

- Addition of the **wait** statement supports the general process modeling. Circuits can be modeled more realistically and easily.
- The IEEE-1076 persistent process model is more general than that of Version 7.2 and is therefore able to model at the system level.
- The simulation model changed because of the addition of the **wait** statement and file I/O allowing more circuits to be modeled.
- The component instantiation definition changed from a primitive to a one defined in terms of block hierarchy.
- Structural recursion using the generate statement is allowed. Because of that, more complex circuitry can be modeled.
- Since signals are allowed in subprograms, new attributes were added for use in subprograms: 'Event, 'Active, 'Last_Event, and 'Last_Active. These should only be used in sequential contexts.
- The simulation time can now be accessed through the predefined function *Now*. This new function gives the ability to simulate mean-time-before-failure and to model Random Access Memory (RAM) decay times.

Conclusion:

More circuits can now be modeled and simulated with less effort than before making IEEE-1076 a more general hardware description language than Version 7.2. All of the above differences have a positive impact on VHDL.

4.6 Conclusion

Based on the evaluation criteria, VHDL has been improved in all categories (criteria). By far, most of the changes have been made to improve the definition of the language. Based on the language criteria and the previous evaluation, IEEE-1076 is an excellent programming language for hardware descriptions. It meets all of the criteria favorably, and should prove to be the best hardware description available. Only time and extensive usage will determine if this conclusion is correct.

4.7 Summary

This chapter described the process of finding predefined programming language criteria the steps followed to perform the evaluation. The details of the evaluation were also presented in this chapter. The next chapter describes the conversion and extension of the Version 7.2 analyzer.

V. Analyzer Conversion-Extension

5.1 Introduction

This chapter details the overall design of the analyzer and the steps performed to convert and extend the analyzer. This task was first described in Chapter 1 and is further elaborated below. An example is also described to aid in understanding the process used.

5.2 Revise and extend the current analyzer.

While performing the evaluation draft and determining what the actual language differences were, the analyzer was revised and extended to conform to IEEE-1076. The VIA had to be redefined for some of the changed constructs as well as the constructs that extended the language. The updated VIA Data Model and Schema are described in [Berk88]. The top-level design of the analyzer was not changed from Bratton's analyzer and is shown in Figure 5.1.

The lexical analyzer which was formed using the UNIX utility *lex* [Lesk78] was changed to accommodate the new reserved words of IEEE-1076. Reserved words from Version 7.2 that are no longer used in IEEE-1076 were deleted.

The parser is still built using *yacc* [John78]. A large portion of the parser was revised to incorporate the numerous changes of the new grammar.

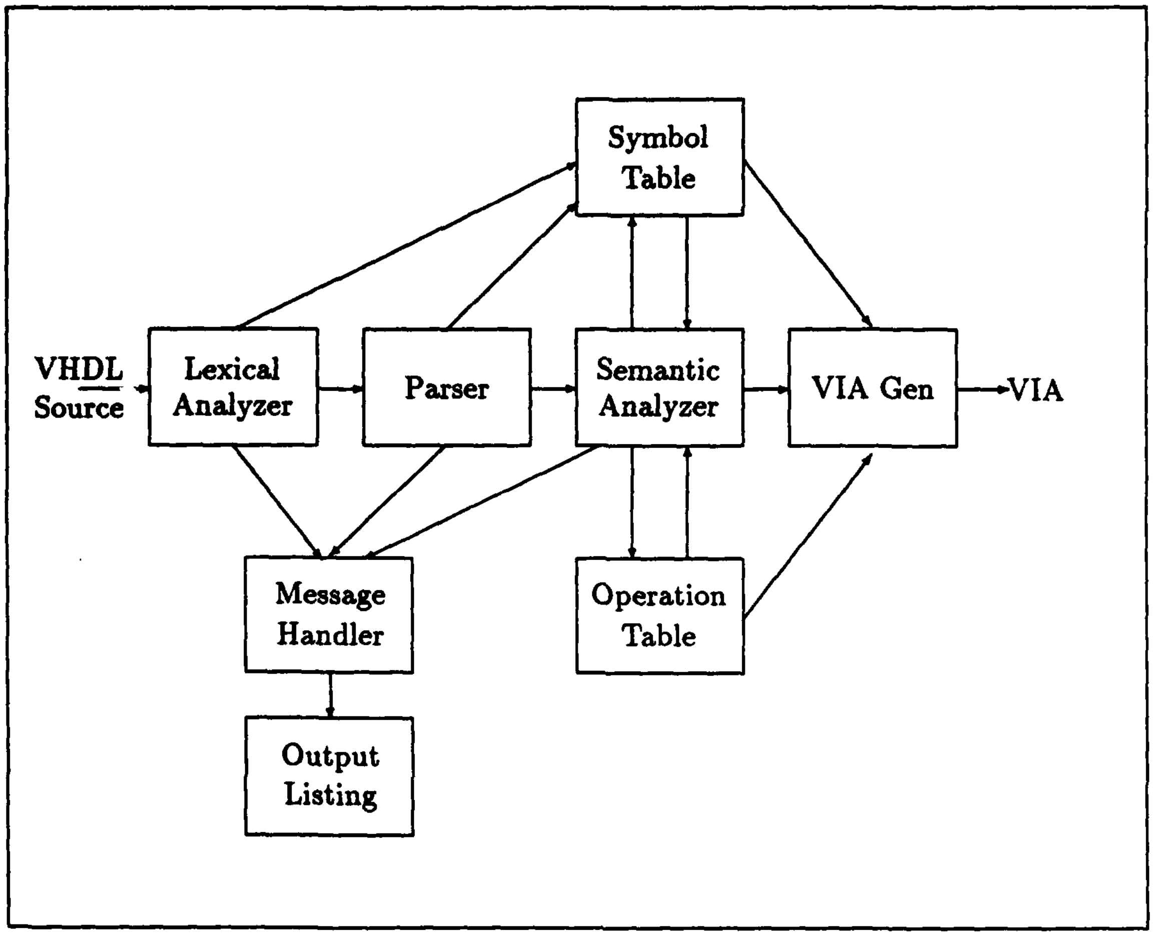

Figure 5.1. Analyzer Design [Brat87]

These changes to the syntax and semantics rippled into the semantic analyzer. The task was made easier since the original design was modular. Many of the modules were changed, and several were added. Great care was taken to preserve the modularity of the design of the analyzer.

The operation table and message handler portions of the analyzer remain unchanged by the new language. The operation table is made up of nodes of an Abstract Syntax Tree (AST). These nodes contain the behavioral portion of the VHDL source code [Brat87]. The message handler section writes messages to the listing output. Messages include fatal, warning, error and "IMP" (not implemented) types.

The symbol table was changed slightly to correspond to the structural changes made to the language. Changes made to the symbol table include those to handle the new structures of subprogram, and the label entry for blocks (to implement the new definition of component instantiation).

The VIA generator changed mostly due to the fact that subprograms are no longer considered design units. A separate category equal to units and objects was created to handle the new definitions of subprograms. The VIA definition was modified and the VIA generator was also changed to match.

Adding language extensions required modifications to the VIA, its generator and the semantic routines once the entire language grammar was incorporated in the the lexical analyzer and the parser. The methods of implementing the changes and extensions are detailed below.

5.3 Conversion-Extension Steps

The following steps describe how the conversion-extension was done:

The first step that was performed was to change the lexical analyzer to recognize and accept all of IEEE-1076. The lexical analyzer was built using the UNIX utility *lex* [Lesk78]. The input file to be processed by *lex* maps sequences of characters to *identifiers* and *literals.* The user defines specific identifiers called reserved words (defined in the IEEE-1076 Language Reference Manual) so that character sequences are recognized as such and cannot be used as variable names. The utility *lex* generates ***yylex*** a C routine from the user defined file to become the lexical analyzer. To perform the lexical analyzer conversion task, two compound delimiters were deleted, and several new reserved words were added while several previous reserved words that are no longer in use were deleted. This procedure was performed only once, and the resulting lexical analyzer was tested to ensure correctness.

The following steps, which are similar to Bratton's implementation [Brat87:4.7], were performed in an iterative fashion using the spiral model described in Chapter 2. Each iteration corresponds to a spiral cycle.

1. Determine the construct to change or extend.
2. Determine which specific validation test cases to use. If none are available, create them.
3. Determine the need for a change or extension to the VIA.
4. Determine if changes are needed to VIA producing code. Design, code and test the changes if necessary.
5. Determine what changes need to be made to the *yacc* file including error recovery.
6. Determine the results of the *yacc* value stack upon completion of parsing.

7. Determine and implement semantic actions that need to occur during the parsing function.

8. Validate construct conversion-extension with both correct and incorrect test cases.

9. Notify the simulator designer [Pomp88] upon successful testing.

As stated earlier in the thesis, the spiral model was chosen because risks are identified upfront, the model was flexible encompassing both software maintenance and new development, and most importantly, problems and errors are detected early. This ensured that each change was implemented correctly before starting the next.

5.4 Conversion Example

To better explain each of the above steps, an example will be shown.

1. *Determine the construct to change.* The construct chosen for this example is *subprograms.* This construct required changes to the VIA as well as the *yacc* file and semantic actions.

2. *Determine validation test cases.* Several test cases were created to show the correct implementation and several were created to show that the implementation detected incorrect input. This was done in advance of the implementation to ensure test cases were available when the implementation was ready for testing and to help understand the changes made to the construct.

3. *Determine if there needs to be a change or extension to VIA.* The VIA already handled subprogram bodies but IEEE-1076 has redefined the role of the subprogram. They are no longer design units so that they had to be removed from that category of VIA. Subprograms now have separate declarations and bodies (to be more Ada-like and to enhance separate compilation) so the VIA must reflect that also. The VIA was restructured to provide a category *subprogram* for the four separate entries: procedure specification (*procspec*), procedure body (*procbody*), function specification (*funcspec*), and function body (*funcbody*). (See Appendix B for details.) Since subprograms are structural in nature these are all symbol table entries.

4. *Determine if changes are needed to the VIA producing code. Design, code, and test necessary changes.* Since the VIA itself was restructured for this change, the code producing the VIA was modified. A new module was also added to detect the *subprogram* category and the four subcategories. This module was designed, coded and then tested.

5. *Determine what changes need to be made to the* yacc *file including error recovery.* In order to do this, the Backus Naur Form (BNF) of the grammar needs close scrutiny. The BNF for the Version 7.2 definition is shown in Figure 5.2 and the BNF for the IEEE-1076 definition is shown in Figure 5.3.

   From these two figures, it can be seen that there doesn't appear to be a large difference between the respective BNF definitions. The main difference is how

```
subprogram_declaration ::=
      subprogram_specification IS subprogram_body;

subprogram_specification ::=
      PROCEDURE identifier [(parameter_interface_list)]
      |FUNCTION identifier [(parameter_interface_list)]
      RETURN type_mark

subprogram_body ::=
      subprogram_declarative_part
      BEGIN
      sequence_of_statements
      END [subprogram_simple_name]
```

Figure 5.2. Version 7.2 Subprogram BNF [Inte85]

```
subprogram_declaration ::=
      subprogram_specification;

subprogram_specification ::=
      PROCEDURE designator [(formal_parameter_list)]
      |FUNCTION designator [(formal_parameter_list)]
      RETURN type_mark

subprogram_body ::=
      subprogram_specification IS
            subprogram_declarative_part
      BEGIN
            subprogram_statement_part
      END [designator]
```

Figure 5.3. IEEE-1076 Subprogram BNF [IEEE88]

the subprogram_specification is defined and used. The Version 7.2 *yacc* file which was based on the BNF definition is shown in Figure 5.4.

Tokens passed by *lex* are upper case names (such as PROCEDURE and FUNCTION), and productions are lower and mixed case names (such as subprogram_specification and .procedure_parameter_list.[1]) The braces in *yacc* denote an action area and contain statements which are executed if the parser accepts the preceeding production. The symbol "$$" represents the *yacc* stack value for the current action statement and the symbols "$1", "$2",...etc, represent the *yacc* stack values for production 1, production 2,...etc, respectively. The *yacc* code was changed to reflect the changes and is shown in Figure 5.5.

6. *Determine the results of the* yacc *value stack upon completion of parsing.* Before any actions were added, the *yacc* value stack for *subprogram_body* was as follows:

   (1) the pointer to symbol table entry for *subprogram_specification*
   (2) the token IS
   (3) the pointer to the symbol table entry for *subprogram_declarative_part*
   (4) the token BEGIN
   (5) the pointer to the Abstract Syntax Tree (AST) node for first node of *sequence_of_statements*
   (6) the token END
   (7) the pointer to the symbol table entry for *.designator.*
   (8) the token Semicolon

[1] *yacc* allows the underscore and period to be used in non-terminals. The naming conventions described by Bratton (Bratton, 1987:4.9) are continued in this project and reiterated for continuity. An optional non-terminal is denoted by a period on either side, a non-terminal that could be repeated one or more times is denoted by two periods on either side, and a list is defined by three beginning periods and two trailing periods.

```
subprogram_declaration
        : procedure_declaration
        | function_declaration

procedure_declaration
        : PROCEDURE
        Identifier
                {
                $$ = s_makeunit(U_PROC, $2);
                }
        .procedure_parameter_list.
        IS
                {
                s_bpush();
                s_setreturn(NULL_SYMBOL);
                }
        subprogram_declarative_part
        BEGIN
        sequence_of_statements
        END_ERR
        .simple_name.
        Semicolon_ERR
                {
        $$ = procedure_declaration($3,$4,$9,$11);
                s_bpop();
                }
        ;

function_declaration
        : FUNCTION
        Identifier
                {
                $$ = s_makeunit(U_FUNC,$2);
                }
        .function_parameter_list.
        RETURN
        type_mark
        IS
                {
                s_bpush();
                s_setreturn($6);
                }
        subprogram_declarative_part
        BEGIN
        sequence_of_statements
        END_ERR
        .simple_name.
        Semicolon_ERR
                {
                $$ = function_declaration($3,$4,$6,
                                $9,$11,$13);
                s_bpop();
                s_setreturn(NULL_SYMBOL);
                }
        ;
```

Figure 5.4. Version 7.2 *yacc* file

```
subprogram_declaration
        : subprogram_specification
        Semicolon_ERR
                {
                s_return(NULL_SYMBOL);
                }
        ;

subprogram_specification
        : PROCEDURE
        designator
                {
        $$ = s_make_subprgmspec(SP_PROCSPEC,$2);
                }
        .procedure_parameter_list.
                {
                $$ = procedure_specification($3,$4);
                }
        | FUNCTION
        designator
                {
        $$ = s_make_subprgmspec(SP_FUNCSPEC,$2);
                }
        .function_parameter_list.
        RETURN
        type_mark
                {
        $$ = function_specification($3,$4,$6);
                s_return($6);
                }
        ;

subprogram_body
        : subprogram_specification
        IS
                {
                s_bpush();
                }
        subprogram_declarative_part
        BEGIN
        sequence_of_statements
        END_ERR
        .designator.
        Semicolon_ERR
                {
                $$ = subprogram_body($1,$4,$6,$8);
                s_bpop();
                s_setreturn(NULL_SYMBOL);
                {
        ;
```

Figure 5.5. IEEE-1076 *yacc* file

For nonterminals, the type is determined by what is returned by the user-defined semantic actions. The integer value of the terminals (tokens) are determined by a list of terminal symbols supplied to *yacc*.

7. *Determine and implement semantic actions that need to occur during the parsing function.* The previous *yacc* file example already had the semantic actions placed in it so they will be explained. Several semantic actions must take place during this analysis. First, a separate declaration region is created by s_bpush() for variables local to the subprogram. The task that *subprogram_body* performs is to link the pointers of the symbol table entries and ASTs (operation table entries) to the symbol table entry created by the *subprogram_specification* when it is parsed. The action s_bpop() clears the declaration area and the *s_setreturn* sets the return flag to NULL_SYMBOL, which is the default. It is only set to a type when a FUNCTION is being parsed.

8. *Validate construct conversion-extension with both correct and incorrect test cases.* The analyzer was tested using correct and incorrect test cases as well as a regression test.

9. *Notify the simulator designer upon successful testing.* When the analyzer had passed its testing, the simulator designer was notified.

5.5 Summary

This chapter presented the overall design and the detailed methods used to implement the design of the analyzer. The steps used to perform the conversion-extension of the analyzer were discussed. An example was shown to help explain the method used to convert the analyzer. Testing of the analyzer to show conformance to IEEE-1076 is presented in the next chapter.

VI. Testing and Analysis

6.1 Introduction

This chapter describes the testing that the analyzer underwent and the test results analysis. It also shows how well the requirements were met that were discussed in Chapter 1. Since the requirements paralleled Bratton's effort [Brat87], the testing was also along the same lines. This provided a convenient comparison and helped to determine if the results were similar. Even though the scope of this project did not include *production quality*, an attempt was made to maintain the quality already built into the analyzer.

6.2 Testing Tasks

The following elaborate the remaining tasks of Chapter 1, those pertaining to testing and documenting.

6.2.1 Develop or obtain a validation test suite for the analyzer. To validate the analyzer for syntactic and semantic conformance, the Intermetrics converted Version 7.2 test suite mentioned in Chapter 1 was used. Additional test cases were built to cover the added construct syntax and semantics. Designs for some TTL circuits written in Version 7.2 were converted to IEEE-1076 to partially form a performance test suite. Other circuit designs and test cases were built as needed.

6.2.2 Validate the changes and extensions. After each set of changes or an extension was implemented, the analyzer was tested using regression testing techniques. This is the suggested method for testing compilers [Aho86:321]. The test suite was run after each implementation to ensure no constructs were inadvertently changed. The results of this testing are found in this chapter and in Chapter 7.

6.2.3 Perform analyzer-simulator integration tests. Once a version or implementation was validated it still needed to be integrated with the AVE simulator. The test cases for this integration testing were the tests that made up the performance test suite and the converted TTL circuits described earlier. This testing showed how well the analyzer performed with the other tools of the AVE as well as such items as resource usage and portability. The results are documented in Chapter 5.

6.2.4 Document the results. This thesis contains the major design decisions, the details of the implementation and the test results for this part of the project.

6.3 Requirements

The requirements for the analyzer were set forth in Chapter 1. This section will briefly review these requirements below.

Generalization

1. Use the common tools of UNIX

2. Be portable

3. Handle both correct and incorrect input

Validation/Reliability

4. Prove constructs were correctly implemented

Maintainability

5. Be well documented
6. Be designed in a structural/modular manner

Economical

7. Conserve computer resources (memory and time)

AVE System Integration

8. Maintain and improve "end-to-end" capability of the analyzer-simulator system

6.4 How the Requirements Were Met

Several types of testing were employed to determine if the requirements were met: conformance testing, performance testing, portability testing, and system integration testing.

Not all the requirements needed to be tested; the maintainability requirement was simply desk checked to ensure modularity.

Conformance testing was used to test requirements 3 and 4, performance testing was used to determine time and memory usage (requirement 7), portability testing was used to show how requirements 1 and 2 were met, and system integration tests were used to show how well the analyzer-simulator system was integrated.

6.5 Testing Process

Each form of testing will be described with an example shown when needed for clarification.

Conformance testing showed how well the analyzer conformed to or met the IEEE-1076 standard. Two types of tests were run. The first type (called short tests) checked if the analyzer could analyze correct input and output the correct VIA. The second type (called error tests) was how well the analyzer picked out deliberate errors in the input source code and how it handled the errors. The correctly coded input tested both the syntax and semantics of the language.

Since the analyzer was converted to handle all IEEE-1076, no syntax errors were allowed to pass without correction either to the analyzer or to the test case if an error was found there and confirmed by consultation with the IEEE-1076 LRM. Semantic errors were allowed for non-implemented constructs.

To continue Chapter 4's implementation example, the following is a test to check the syntax in its simplest form. To test for a subprogram declaration:

```
entity A1 is
  procedure one;
  function two return boolean;
end A1;
```

To check subprogram bodies the following architecture was appended to the above entity:

```
architecture test of A1 is

  procedure one is
  begin
  end one;

  function two return boolean is
    variable I : boolean;
  begin
    I := TRUE;
  Return I;
  end two;
  begin
end test;
```

The above tests would also check the semantic requirement of an entity name being declared and analyzed before the architecture body. The results of the analysis (when the verbose option is employed) is shown below.

```
AFIT VHDL Analyzer Revision: 1076 - 3.1
[1] entity A1 is
[2]   procedure one;
[3]   function two return boolean;
[4]  end A1;
[5]
[6]  architecture test of A1 is
[7]
[8]   procedure one is
[9]   begin
[10]   end one;
[11]
[12]   function two return boolean is
[13]   variable I : boolean;
[14]   begin
[15]   I := TRUE;
[16]   return I;
[17]   end two;
[18]  begin
[19]  end test;
[20]
[21]
Number of errors detected: 0
Generating VIA file...done
VHDL Analysis complete.
```

The results of the conformance testing are found in Table 6.1.

Table 6.1. **Results of Test Suite Testing**

Result	Short		Error		Total	
Pass	127	(56%)	217	(67%)	344	(61%)
Fail	18	(8%)	33	(10%)	51	(09%)
Not Impl	81	(36%)	73	(23%)	154	(28%)
Total	226		409		549	

The test suite actually contained 710 test cases. However, 161 of these test cases were unable to be used because of errors such as a package body without its corresponding package declaration. These test cases were not analyzable. The maintainer of the test suite has been notified.

For *portability testing* the analyzer files were moved to a SUN Work Station operating under UNIX BSD Version 4.2. The analyzer was developed on an ELXSI 6400 using UNIX BSD Version 4.2 without using any system specific calls. The analyzer was completely recompiled on the Sun and another ELXSI operating under UNIX BSD Version 4.3. It was then tested using a dataflow and a procedural design capturing the time to run using the UNIX *Time* command. The results are found in Table 6.2. The computers were lightly loaded at the time of testing.

Table 6.2. **Portability Tests**

Design Type	Processor	
	SUN	ELXSI
Dataflow	0.23[1] (0.08)[2]	0.12 (0.04)
Procedural	0.24 (0.05)	0.11 (0.03)
1. Average CPU seconds 2. Std dev		

Performance testing was performed in concert with the portability testing. The UNIX *time* command was used to determine the time to run certain designs on each of the computers. To determine how much memory was used, the *time* command again was used. Less than 600k bytes were needed to analyze each test case in the validation test suite. Also included in the performance testing was the size of the

VIA file when compared to the number of lines of source input. This was found simply by dividing the size of the VIA file (bytes) by the number of lines of source VHDL (comments were not counted). The average VIA file from the validation test suite was 2855 bytes and the average lines of code from each input file was 28 lines. The VIA file included the overhead of package Standard each time since the analyzer does not incorporate a design library at this time.

Integration testing was performed by using several circuits that were meaningful. Each circuit was analyzed and simulated to show that indeed the AVE tools were integrated.

6.6 Summary

This chapter described the types of testing the analyzer was subjected to and the results of those test. The final chapter, Chapter 7 show the conclusions of the project and future recommended research.

VII. Conclusions and Recommendations

7.1 Introduction

This thesis presented an evaluation of the changes and additions to Version 7.2 that formed IEEE-1076. This project also converted and extended a current Version 7.2 subset analyzer. Future research, which is recommended below, will further enhance the current analyzer, the AVE, and the knowledge of the VHDL language itself.

7.2 Conclusions

7.2.1 **Evaluation** The evaluation of the two versions of VHDL proved to be valuable. It not only aided in understanding the differences between the languages, it also provided insights into the subtle differences that are not intuitively obvious. This knowledge was useful when performing the actual conversion of the analyzer.

The evaluation reveals that VHDL has been significantly improved in its current form, IEEE-1076. Most noticeably, the language has become better defined as evidenced by the number of changes (39) falling under the *well-definedness* criterion. Consistency of the syntax will have a positive affect on the users of VHDL.

The VHDL language has become a clearer more consistent language. New features added to the language such as File IO makes the language more powerful. The addition of a dynamic data type also allows modeling of more circuitry than before. The language remains high-level enough to preclude being bound to any one computer architecture. The language remains very modular and is quite extensible with subprogram and operator overloading. VHDL is also more geared toward team design than before.

IEEE-1076 improved VHDL in all criteria used in the evaluation.

7.2.2 **Conversion/Extension**

The conversion and extension of the subset analyzer provided a more robust analyzer than the one at the beginning of the project. Adding the *wait statement* and *component instantiation* allowed greater simulation capabilities for "end-to-end" results (analyzer-simulator)

Conversion of the analyzer, however, was not as trivial as first thought. Providing separate declarations and bodies for subprograms and packages proved to be the first major hurdle. Several new ambiguities were introduced in IEEE-1076 since part of the grammar is context dependent, causing many reduce/reduce conflicts which had to be resolved in the *yacc* parser.

The results of the testing described in Chapter 5 showed that the following requirements were met.

1. The Analyzer runs on different computers with different versions of UNIX installed.
2. The Analyzer was tested thoroughly.
3. The Analyzer maintains its modularity and is well- documented.
4. The Analyzer is conservative in it use of memory and time.
5. The Analyzer was tested in an integrated environment with the AVE Simulator.

7.3 Recommendations for Future Research

7.3.1 Continue to Study the Language. Since IEEE-1076 is still very new and few tools are available to use it, not much is really known about how much better it is in practice than Version 7.2. Studying certain circuits and programming them in both Version 7.2 and IEEE-1076 would be beneficial to other users.

7.3.2 Complete Implementation of the Language. The analyzer does not yet implement the whole language of IEEE-1076. It would be beneficial for further research into the different ways of implementation of IEEE-1076 constructs into such areas as memory and speed optimization. Once the entire language is implemented and validated, this would be an area for yet further study.

7.3.3 Design Library Implementation. The AVE currently does not support a design library. All structures must be in the source file and analyzed together. To provide efficient use of computing resources, separate compilation and a library manager are needed to support team development for large circuits. Resources can be conserved by analyzing only those parts or packages that have changed.

7.3.4 Determine a Suitable Subset of VHDL. Steps are now being taken to define a subset of VHDL (Weiss, 1988: 51) since some vendors are having great difficulty interfacing to the whole language. Some of the problems with defining subsets are what to include and how to control the proliferation of subsets. Future research might try to define specific subsets and justify why those particular constructs should be grouped together. Transportability between subsets should also be addressed.

7.4 Summary

The evaluation of the two language versions showed that IEEE- 1076 is an improvement over Version 7.2 according to the criteria used in the evaluation. This gives the VHDL designer more capabilities and a language that is easier to use and understand. The VHDL analyzer produced during this research provided a basis for understanding VHDL-1076 and a tool that is suitable for use on UNIX-based computers. Any benefit this ongoing research provides to the academic community and industry benefits the Air Force.

Bibliography

[Aho86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley Publishing Co., 1986.

[Arms88] Armstrong, J.R. "Chip-Level Modeling With HDLs," *IEEE Design & Test of Computers*, *5* (1): 8-18 (February 1988).

[ASD83] Aeronautical Systems Division (ASD), Air Force Systems Command. *VHSIC Hardware Description Language (VHDL) Program.* Solicitation No. F33615-83-R-1003. Wright- Patterson AFB OH, 30 March 1983.

[Aylo86] Aylor, J. H. and others. "VHDL-Feature Description and Analysis," *IEEE Design & Test of Computers*, *3* (2): 17-27 (April 1986).

[Berk88] Berk, Capt Kevin J. and Maj Joseph W. DeGroat. *VHDL Intermediate Access Data Model and Schema.* AFIT-TR-EN-88-OV1. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, November 1988.

[Boeh88] Boehm, Barry. "A Spiral Model of Software Development and Enhancement", *Computer*, *10* (5): 61-72 (May 1988).

[Brat87] Bratton, Capt Randolph M. *A Production-Quality UNIX Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) Subset Analyzer.* MS Thesis AFIT/GCS/MA/87D-1. School of Engineering, Air Force Institute of Technology (AU), Wright- Patterson, AFB OH, December 1987.

[CLSI87] CAD Language Systems, Inc. (CLSI). *VHDL Language Refinement Rationale.* Rockville MD, August 1987.

[Cart88] Carter, Dr Harold W. Telephone interview. Cincinnati OH, 24 May 1988.

[Cart87] Carter, LtCol Harold W. and others. *1986 Research Report: AFIT VHDL/DB/DBMS Research.* AFIT-ENC-TR-87-01. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, January 1987.

[DeGr88] DeGroat, J. W. and others. "The AFIT VHDL Environment," *Proceedings of the IEEE 1988 Frontiers in Education Conference.* pg 324. New York: IEEE Press, 1988.

[Deit84] Deitel, Harvey M. *An Introduction to Operating Systems.* Reading MA: Addison-Wesley Publishing Company 1984.

[DoD88] Department of Defense. *Military Standard Defense System Software Development.* DOD-STD-2167A. Washington: Department of Defense, 26 February 1988.

[Dewe86] Dewey, Allen and Anthony Gadient. "VHDL Motivation," *IEEE Design & Test of Computers*, *3* (2): 12-16 (April 1986).

[Frau86] Frauenfelder, Capt Deborah J. *An Implementation of a Language Analyzer for the Very High Speed Integrated Circuit Hardware Description Language.* MS Thesis AFIT/GCE/MA/86D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986 (AD-A178648).

[Gilm86] Gilman, Alfred S. "VHDL-The Designer Environment," *IEEE Design & Test of Computers, 3* (2): 42-47 (April 1986).

[IEEE88] IEEE. *IEEE Standard VHDL Language Reference Manual.* IEEE Std 1076-1987, The Institute of Electrical and Electronics Engineers, Inc., New York NY, 1988.

[Int84a] Intermetrics, Inc. *VHDL User's Manual: Volume II - Usage Scenarios.* Contract F33615-83-C-1003. Bethesda MD, 30 July 1984.

[Int84b] ——. *VHDL Analyzer Test Plan.* Contract F33615-83-C-1003. Bethesda MD, 30 July 1984.

[Inte85] ——. *VHDL Language Reference Manual Version 7.2.* Contract F33615-83-C-1003. Bethesda MD, 1 August 1985.

[John78] Johnson, S. C. "Yacc: Yet Another Compiler-Compiler," Murray Hill NJ: AT&T Bell Laboratories, 31 July 1978.

[Kern78] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs NJ: Prentice- Hall, Inc., 1978.

[Kern88] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language* (Second Edition). Englewood Cliffs NJ: Prentice-Hall, Inc., 1988.

[Knap84] Knapp, David W. and Alice C. Parker. *A Data Structure For VLSI Synthesis and Verification.* Contract DAAG29-80-K-0083. Department of Electrical Engineering- Systems, University of Southern California, Los Angeles CA, 8 May 1984.

[Lesk78] Lesk, M. E. and E. Schmidt. "Lex—A Lexical Analyzer Generator," Murray Hill, NY: Bell Laboratories, 31 July 1978.

[Lips86] Lipsett, Roger and others. "VHDL-The Language," *IEEE Design & Test of Computers, 3* (2): 28-41 (April 1986).

[Lowe86] Lowenstein, Al and Greg Winter. "VHDL's Impact on Test," *IEEE Design & Test of Computers, 3* (2): 48-53 (April 1986).

[Mear88] Mears, L.A. and Steven Pl Levitan. "An Integrated Capture and Simulation Tool for Digital Designs," *Spring 1988 DARPA Microsystems Design and Prototyping Contractors Meeting.* April 1988.

[Nash86] Nash, J.D. and L. F. Saunders. "VHDL Critique," *IEEE Design & Test of Computers, 3* (2): 54-65 (April 1986).

[Pomp88] Pompilio, Capt Douglas. *UNIX-Based IEEE-1076 VHDL Simulator.* MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

[Pres87] Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* (Second Edition), New York: McGraw-Hill Book Company, 1987.

[Radk88] Radke, Charles. "A D&T Roundtable Design Automation Standards," *IEEE Design & Test of Computers, 5* (1): 48-55 (February 1988).

[Schr85] Schreiner, A.V., and H.G. Friedman, Jr. *Introduction to Compiler Construction with UNIX.* Englewood Cliffs: Prentice-Hall, Inc., 1985.

[STEE78] "STEELMAN: Requirements for High Order Computer Programming Languages," Department of Defense, Washington, D.C., 1978.

[Tuck86] Tucker, Allen B. *Programming Languages* (Second Edition). New York: McGraw-Hill Book Company, 1986.

[Waxm86] Waxman, Ron. "The VHSIC Hardware Description Language - A Glimpse of the Future," *IEEE Design & Test of Computers, 3* (2): 10-11 (April 1986).

[Weis88] Weiss, Ray. "VHDL Subsets for CAE," *Electronic Engineering Times.* Issue 497: 51,62 (August 1, 1988).

Vita

Captain Kevin J. Berk was born on [redacted] He graduated from St. Joseph High School in 1973, and in the Fall of 1973, he enlisted in the Air Force as a Special Electronics Technician. He was stationed in the Philippine Islands and at Patrick AFB, Florida. Near the end of his enlistment, he received an AFROTC scholarship and attended the University of Central Florida where he pursued a Bachelor of Science in Physics. He was also selected as an AFROTC distinguished graduate and commissioned in the Air Force. After his return to active duty, he served as a project officer at the Shuttle Test Group at Vandenberg AFB, California until he was selected for the Undergraduate Engineering Conversion Program. In September 1982 he entered Michigan State University pursuing a second Bachelors degree in Electrical Engineering. After graduating with honors, he was assigned to the Aeronautical Systems Division, Wright-Patterson AFB, Ohio where he served as an Embedded Computer Resources Engineer. He entered the School of Engineering, Air Force Institute of Technology, in June 1987 to pursue a Masters of Science in Computer Engineering.

[redacted]

UNCLASSIFIED

~~SECURITY CLASSIFICATION OF THIS PAGE~~ A202 739

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/88D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering; 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFWAL; 8b. OFFICE SYMBOL *(If applicable)* AAD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Air Force Wright Aeronautical Laboratory Wright Patterson AFB 45433-6503	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
THE IMPACT OF IEEE-1076 ON VHDL (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Kevin J. Berk, B.S., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MSCE Thesis	FROM ______ TO ______	1988 December	76

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Computer Programs, Compilers, Programming Languages, Computer Aided Design, VHDL, —(THESES)—([illegible])
12	05		

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Approved for release in Accordance with AFR 190-1 [illegible]

Thesis Chairman: Maj Joseph W. DeGroat

This paper describes the conversion and extension of the Air Force Institute of Technology's (AFIT's) UNIX-based VHDL Analyzer. This task concentrated on converting the current AFIT analyzer from VHDL Version 7.2 to IEEE Standard 1076-1987 and extending it to include the wait statement and component instantiation as defined in IEEE Standard 1076-1987. An evaluation was also performed on the differences between Version 7.2 and IEEE Standard 1076-1987 using nine predefined criteria that determine if a programming language is good. The evaluation was done to determine if IEEE Standard 1076-1987 was indeed a better version of VHDL than its predecessor, Version 7.2. — [illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION	
22a. NAME OF RESPONSIBLE INDIVIDUAL Kevin J. Berk, Captain, USAF	22b. TELEPHONE *(Include Area Code)* (513) 255-3030	22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED